What are coroutines?

- Suspendable, “interruptable” functions
- Extension of Kotlin in the version 1.1
  - New Keyword “suspend”
  - Coroutines are currently “experimental”
- Alternative for Threads
  - 100’000 Coroutinen are no problem
- Are converted to statemachines by the compiler

Interruptable functions?

- Not normal functions
  → Only callable with special library functions

```
val singleValue: Deferred<Int> = async { 1 }
val result: Int = runBlocking { singleValue.await() }
result == 1
```

- Multiple results

```
val manyValues: Sequence<Int> = buildSequence {
    var v = 0
    while(true) {
        yield(v)
        v += 1
    }
}
```

Coroutine Builders

- *’runBlocking’* and *‘buildSequence’* are coroutine builders
- Define the kind of coroutine
  - z.B. ‘*buildSequence*’ generates a lazy Sequence - ‘*yield*’ is part of the ‘*buildSequence*’ builder
- More builders exist
  - ‘*launch*’
  - ‘*async*’
  - …

Infinite sequence

```
val infiniteValues = buildSequence {
    var v = 0
    while(true) {
        yield(v)
        v += 1
    }
}
```

```
infiniteValues.forEach { value ->
    println("Currently $value")
}
```

Alternative for callbacks

```kotlin
fun needCallback(input: Int, resultCB: (String) -> Unit) {
    val result = input.toString() // time consuming...
    resultCB(result)
}
…
needCallback(10, { x -> println("x is $x")})
```

```kotlin
suspend fun noNeed(input: Int) = suspendCoroutine<String>{ continuation ->
    val result = input.toString() // time consuming...
    continuation.resume(result)
}
…
val x:String = noNeed(10)
println("x is $x")
```

Coroutine as sequence of functions

```
suspend fun sequence() {
    val a = async {
        Thread.sleep(1000)
        6
    }.await()

    val b = async {
        Thread.sleep(2000)
        7
    }.await()

    println(”Answer: ${a ∗ b}")
}
```

```
fun part1(then: (Int) -> Unit) {
  Thread.sleep(1000)
  then(6)
}
fun part2(then: (Int) -> Unit) {
  Thread.sleep(2000)
  then(7)
}
fun sequence() {
  part1 { a ->
    part2 { b ->
      println(
        "The answer is ${a + b}”
      )
    }
  }
}
```

Continuation passing style - CPS

Keyword “suspend” implies a hidden parameter – the continuation:

```
suspend fun sequence() { … }
```

```
fun sequence(c: Continuation<Unit>) {
    …
    UI_ThreadPool.submit(c)
}
```

Coroutine Context

- Thread selection

```
launch(UI) {
    sequence()
}
```

→ Führt die Coroutine auf dem UI Thread aus

- Access on “coroutine-local” variables

```
class AuthUser(val name: String) :
AbstractCoroutineContextElement(AuthUser) {
    companion object Key : CoroutineContext.Key<AuthUser>
}
…
async(UI + AuthUser("me")) {
    val user = coroutineContext[AuthUser]?.name
}
```

Advantages of coroutines

- Small resource usage
- Have the appearance of functions, but are statemachines
  → small overhead
- Allows asynchronous, imperatives programming
  → all constructs work as usual:
  - Try-catch, try-with-resources
  - Loops
  - Etc.
- Alternative for threads

Disadantages of coroutines

- Status is experimential in Kotilin 1.1 → not yet part of the language
- Code looks linear, but its not

Example: Login Prozess

```
Handler().postDelayed({
    object : AsyncTask<Void, Void, Void>() {
        public override fun doInBackground(vararg voids: Void): Void? {
        …
        }
    }.doInBackground()
}, 1000)
```

↓

```
launch(UI) {
    delay(1000, TimeUnit.MILLISECONDS)
    …
}
```

Example: Login blockieren

```kotlin
suspend fun verifyPassword(password: String, userEmail: String) : Boolean
{
    val passwordValid = password == "123456"
    if(!passwordValid) {
        delay(10, TimeUnit.SECONDS)
    }
    return passwordValid
}
```